React JS

What is react?

* It’s a library not a framework
* Focus only on UI, don’t focus on http calls/routings, only responsible for rich Ui
* Has rich ecosystem for other purposes.
* Its component base architecture.
* Can write reusable code, that component can be used in angular or vue.

Declarative Paradigm: just tell what to do, how it does is up to them.

Imperative paradigm: we explicitly say what to done step to step, we control the flow.

* React is declarative (tell what you want it build actual UI). Declarative paradigm in contrast to imperative paradigm we use to which implements algorithm explicit steps.
* We have to what to create react will create actual UI.
* It handles efficiently and updates and renders the exact component and updates the DOM.
* It can be integrated to any app as a portion or complete page or whole application.
* React native is used for mobile apps.
* Html, CSS, JavaScript, ES6,
* In JavaScript – ‘this’ keyword, filter, map and reduce.
* In ES6 – let & const, arrow functions, template literals, default parameters, object literals, rest and spread operators and DE-structuring assignment.

Folder structure

* We want react to control the app for that we have one div tag with id = root one time react app will take control and it responsible rendering. (this is called as root DOM node)
* This root is tagged to index.js file with that elementById(‘root’).

Component structure:

* Components can contain many components ex: app component (acts as parent component)
* In react we have two types of components:

1. Stateless Functional component
2. Stateful class component

|  |  |
| --- | --- |
| Stateless functional component | Stateful class component |
| 1. Are literally JS functions they return html which describe the UI. | Class extending component class.  Render method returning HTML |

Functional vs Class Components

|  |  |
| --- | --- |
| Functional component | Class component |
| 1. Simple functions receiving props and returning declaration. 2. Use functional component as much as possible. 3. Absence of ‘this’ keyword. 4. Solution without using state 5. Mainly responsible for the UI 6. Also called as Stateless/Dumb/Presentational | 1. More feature rich 2. Maintain their own private data – state 3. Complex UI logic 4. Provide lifecycle hooks 5. Stateful/Smart/Container components |

**JSX (JavaScript XML)**

Write XML-like code for elements and components.

JSX tags have a tag name, attributes and children.

JSX makes code simpler and elegant. (Not necessity to use JSX)

JSX ultimately transpiles to pure JS which is understood by the browser.

**JSX differences**

Class -> className

For -> htmlFor

CamelCase property

* onclick -> onClick
* tabindex -> tabIndex

**Props**: are immutable

React components should work as pure function with respect to props parameter cannot change at any time.

All the application make use of props.

**Props VS State**

|  |  |
| --- | --- |
| Functional component | Class component |
| 1. props get passed to the components 2. Functional parameters 3. props are immutable 4. Accessed as props – Functional Components 5. Accessed as useState Hook – Functional Components | 1. State is managed within the component 2. Variables declared with in the function body 3. It is managed within the component so it has full control 4. Accessed as this.props – Class Components 5. Accessed as this.state – Class Components |

**Do’s and DoNot’s**

* We should never modify **state** directly. Use **setState**.
* Whenever you want to execute code after state has changed don’t place codes after **setState** method instead place that in callback function, and pass it as a second parameter for **setState** method.
* **this**.setState(

{count: **this**.state.count + 1},  
 () => {console.log('from call back', **this**.state.count)}

);

* React group multiple **setState** call to single update for better performance.
* Whenever you want to update state base on the previous state make sure to pass function as an argument instead of passing regular object.
* **this**.setState((prevState, props) => ({  
   count: prevState.count + props.index  
  }), () => {  
   console.log('from callback', **this**.state.count)  
  });
* Only place we can assign state is constructor, any other place if you want to update state you have to use **setState** method.
* Call to **setState** are asynchronies, if you place next to setState it will not update.

**Destructuring Props and State:**

* Destructuring in the parameter.
* **const** Greet = ({name, greet, children}) => {  
   **return** (<div><h1>Hello {name} {greet}</h1>{children}</div>)  
  };
* Destructuring in function body.
* **const** Greet = props => {  
   **const** {name, greet, children} = props;  
   **return** (<div><h1>Hello {name} {greet}</h1>{children}</div>)  
  };
* You can extract necessary props form the props object instead of all.

**Event Handling**

* Ex: onClick={clickHandler}
* We should not use as a function call Ex: onClick={clickHandler()} this will get trigged when the component get rendered. This comes worst in class component.
* In class component Ex: onClick={this.clickHandler}

**Bind Events**

1. Binding in the render method. Ex: onClick={this.clickHandler.bind(this)} this is the default functionality of JS by using bind(this).
2. Arrow function approach in render method, using Ex: onClick={() => this.clickHandler()} in this () are required. \*\*If you want to pass parameters use this approach.
3. Binding in the constructor, this the official approach in react docs. This is better than render method. Ex: this.clickHandler = this.clickHandler.bind(this)
4. Use arrow function as class property it’s the way you define the method.

**Parent Child Components Communication** (methods as props)

* We use props to pass any data from parent to child component.
* To communicate child to parent we still use props but pass in reference to method as props to the child component.
* In the child component use props object to access the method in parent component.
* If you want to pass parameter form the child to parent use arrow functions.

**Conditional Rendering**  Ex: UserGreeting.js

1. If/else

* If/else do not work inside **JSX** because JSX is a syntax sugar for function calls and object construction.

Ex: **if** (**this**.state.isLoggedIn) {**return** <div>Welcome srini</div>}  
**else** {**return** <div>Welcome Guest</div>}

1. Element variables

* Using temporary variable and return the value.

**Ex: let** message;  
**if** (**this**.state.isLoggedIn) {message = <div>Welcome srini</div>}   
**else** {message = <div>Welcome Guest</div>}  
**return** <div>{message}</div>

1. Ternary conditional operator

Ex: **this**.state.isLoggedIn ? <div>Welcome srini</div> : <div>Welcome Guest</div>

1. Short circuit operator

Ex: **return this**.state.isLoggedIn && <div>Welcome srini</div>

**List Rendering**

* Rendering list of data using map() as we in JavaScript.

Ex: **const** names = [ 'sri', 'srini', 'srinivas'];  
 **const** nameList = names.map(name => <h2>{name}</h2>);  
 **return**(<div>{nameList}</div>)

* Simplify as much as possible. Ex: NameList.js and Person.js
* **Key** is a special string attribute you need to include when creating list of elements.
* **Keys** give the elements a stable identity.
* **Keys** help react to identify which items have changed or added or removed to help the efficient update the **UI.**
* Each item render in the list by using map must have a **prop** called **key** and the **value** the **prop** must be **unique** with in the **list.**
* **Key** prop is **reserved** not accessible in the child component it’s only used to render the list efficiently, if you want the access same value within the child component you should pass it as a different prop.
* While there is an update in the lists React renders both at the same time and generates mutation whenever there is difference it simply **insert** the item in to the **DOM** tree instead for clearing old tree and construct new tree.
* Without **key** props React end up thinking the entire list items are different this is **inefficient** way of doing and reduce the **performance**.

**Index as Key**

* When to use **index** as a **key?**
  1. The items in the list do not have a unique id.
  2. The list is a static list and will not change.
  3. The list will never be reordered or filtered.
* To avoid using **index** as **key**
  1. Try one of **npm** packages or hashing one the unique value from one the existing properties
* **Try to avoid using index as key.** In fact react uses index as a key if we not specify the key.

**Styling React Components**

1. CSS style sheets

**In js file**

**function** Stylesheet(props) {  
 **let** className = props.primary ? 'primary' : '';  
 **return**(<div><h1 className={`${className} size-xl`}>Stylesheet</h1></div>)  
}

**In CSS file**

.primary{color: orange;}  
.size-xl{font-size: 50px;}

1. Inline styling

**const** heading ={fontSize: '40px', color: 'blue'};  
**function** Inline() {  
 **return**(<div><h1 style={heading}>Inline</h1></div>)  
}

1. CSS Modules

* This are from CSS modules, the file name Ex: appStyles.module.css
* This is the best approach of using CSS because we reference styles to a local variable it can’t be used in the children component.

In js file

* import styles from './appStyles.module.css';
* <h1 className={styles.success}>success</h1>

In CSS file

* .success {color: green;}

1. CSS in JS Libaries

**Form Handling**

1. Form elements whose values are controlled by react is called **controlled component.**
2. First initial value is set from the state and propagating changed value to state and back to field, react always have the access to component state which reflects the updated values of the form elements that state object can be used to submit the form data when needed.
3. Whenever there is a change in **onChange={}** method is triggered and updates the state in a cyclic process.
4. To prevent reloading the page and emptying the form we have to use **preventDefault()** method.

.preventDefault()

1. It always better to use **onSubmit={this.handleSubmit}** in the **form** button type=”submit”.

**Lifecycle Methods**

1. React provides us built in methods which we can override at particular stages in the lifecycle if needed.
2. This are available only in class components don’t exist in functional components.
3. We can classify in to 4 types

* Mounting: when an instance of a component is being created and inserted into the DOM, It has 4 mounting methods:
* Constructor, static getDerivedStateFromProps, render and componentDidMount.
* Updating: When the component is being re-rendered as a result of changes to either props or state. It has 5 update methods, 3 of them are rarely used methods.
* Static getDerivedStateFromProps, shouldComponentUpdate, render, getSnapshotBeforeUpdate and componentDidUpdate.
* Unmounting: when a component is remove from the DOM.
* Error handling: when there is an error during rendering, in a lifecycle method or in the constructor of any child component.

**Mounting Lifecycle Methods**

1. **Constructor (props)**

* This is call whenever new component is created.
* This is perfect for initializing state and binding the event handlers.
* Never **call** **HTTP requests** form constructor it causes side effects like making ajax calls.
* We have to call special function **super(props)** this will call base class constructor.
* We have access to **props** only after calling **super()** after passing props as an argument.
* This only place you can set or change the state directly by overwriting **this.state** fields in all other cases you to make use of **this.setState().**

1. **Static getDerivedStateFromProps(props, State)**

* This is very rarely used lifecycle method.
* This many used when the state depends on the changes in props over time.
* Ex: we have a component but the initial state depends on the props we pass to the component.
* As this is a static method we can’t make use of **this key word** we can’t call **this.setState()** instead you can simply an object that represent the new state of the component.
* Do not cause side effects, Ex: HTTP requests.

1. **Render()**

* This is required method in the class component.
* We simply read props & state and return JSX which describe the UI.
* This is a pure function it should render for the give props and state.
* Don’t change state or interact with DOM or make ajax calls.
* Right after parent **render()** Children components lifecycles method are also executed.

1. **componentDidMount()**

* This is invoked immediately after all its child components has rendered to DOM.
* Cause side effects. Ex: Interacting with DOM or perform any ajax or network calls to load data.

**Updating Lifecycle Methods**

* This methods are called whenever component are re-rendered because of change in **props** or **state**.

1. **Static getDerivedStateFromProps(props, State)**

* This method is called every time component is re-rendered, this is one of the rarely used method.
* It has receive **props & state** as parameter has to return null or object that represent the update state.
* This is used when the state depends on the props of the components.
* Don’t cause side effects like making HTTP requests.

1. **shouldComponentUpdate(nextProps, nextState)**

* This mainly used for performance optimization, this one of the rarely used method
* It has receive updated **props & state** as parameter and returning **true** or **false.**
* By default every class component re-renders when they receives props or state changes this method can **prevent** default behavior by returning **true** or **false.**
* In this method we can compare the existing props & state values to next props & state values and returning **true** or **false** and let react whether component to update or not.
* Don’t cause side effects like making HTTP requests.

1. **Render()**

* We simply read props & state and return JSX which describe the UI.
* Do not change state or interact with DOM or making ajax calls.

1. **getSnapshotBeforeUpdate(prevProps, prevState)**

* This method accepts prevProps and prevState as parameters is called right before the changes from the virtual DOM are to be reflected in the DOM.
* Use this method to capture some information from the DOM. Ex: we can read the user scroll position and after the update we can maintain that position by performing some calculation.
* This method will either return null or return a value. Returned value will is passed as third parameter to the next method **(componentDidUpdate(prevProps, prevState, snapshot)**.
* This is rarely used method.

1. **componentDidUpdate(prevProps, prevState, snapshot)**

* This method is called after the render is finished in the re-render cycles.
* You can make sure all the components and sub-components are properly rendered itself after the update.
* It accepts 3 parameters **prevProps, prevState, snapshot** value returned from the **getSnapshotBeforeUpdate** method.
* This method is guaranteed to be called only once in re-render cycle.
* Cause side effects i.e. you can make ajax calls, but before making a call you need to compare the **prevProps** and **current Props** then decide whether to make ajax call or not. If you are not comparing you are making unnecessary call.

**Unmounting LifeCycle methods**

1. **componentWillUnmount()**

* This method is invoked immediately before a component is unmounted and destroyed.
* We can perform cleanup tasks cancelling any network requests removing events handlers, cancelling any subscriptions and also invalidating timers from setTimeOut or setIntervel.
* Do not call the setState method.

**Error Handling Lifecycle methods**

* This has two methods.

1. **Static getDerivedStateFromError(error)**
2. **componentDidCatch(error, info)**

* This methods are called when there is an error either during rendering, in a lifecycle method, or in the constructor of any child component.

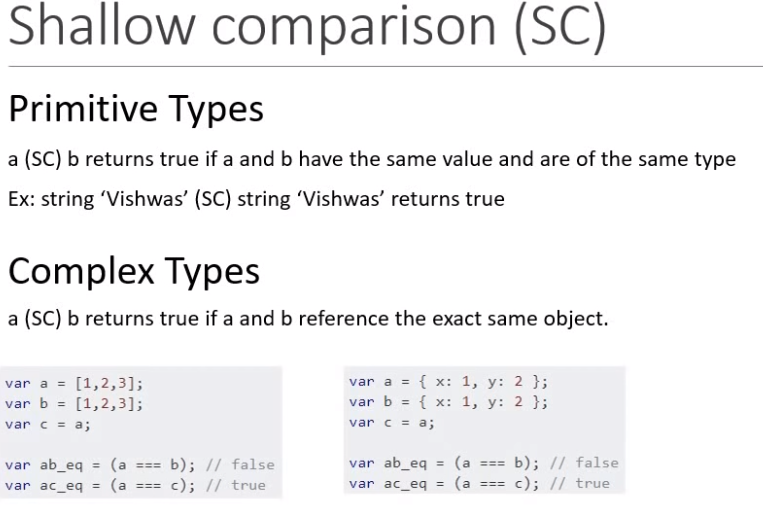
**Fragments**

1. It is used re-place of an extra node element tags while wrapping the multiple elements in JSX.
2. It’s similar to <ng-container> in Angular.
3. **Key** is only attribute we can pass in **React.Fragment.**
4. Short hand syntax is **<><h1></h1><p></p></>** but we can pass **key** attribute.
5. Allow you group list of children elements without adding extra node elements to the DOM.

**Pure Components**

|  |  |
| --- | --- |
| **Regular Component** | **Pure Component** |
| A regular component does not implement the method shouldComponentUpdate method.  It always return true by default. | A pure component implements method shouldComponentUpdate with a **shallow** **props** **and** **state** **comparison**. |

* SC of prevState with currentState, SC of prevProps with currentState if there is a difference then the pure component re-renders.
* Pure components are used avoid the unnecessary rendering and improve the performance boost.
* Don’t **mutate** items in to the list by pushing in to the list because the reference to the array never changes and the pure component will only check for that **never re-render** even if there is a difference always **return a new array or object** when dealing with pure components.
* It’s a good idea to ensure that all the children components are also pure to avoid unexpected behavior.
* Never mutate the state, always return new object that reflects the new state.
* It’s always better to use **regular** components unless you hit with **performance** hit.
* This only works with **class components** don’t work with **functional components.**



**React Memo**

* This work in similar to pure components in functional components.
* You have to export default as below.

Ex: export default React.memo(MemoComp);

**Refs**

* Refs make possible to access DOM node directly with in react.
* It’s possible to pass **ref** from parent to child component.
* Refs can be used in **class** component, **can’t** be attached to **functional** components.

|  |  |
| --- | --- |
| **CreateRef** | **Call back** |
| * With createRef method we use as below * this.inputRef = React.createRef() * In createRef approach we uses **ref** prop and assign the property. * Here the element can be accessed by   this.inputRef.current.focus(); | * In the call ref approach first create property and create method that assigns the property with DOM element passed as parameter. * In the call back approach we attach the ref to an element using the method that intern assign the element to the property. * It is directly accessed by   this.cbRef.focus() |

**Forwarding Refs**

* This is technic for automatically passing a **ref** through a component to one of it children.
* Ex: FRParent.js and FRInput.js
* This is rarely used.

**React Portals**

* This provide a way to render children to DOM node that exist outside the DOM higher key of the parent component.
* This provide the ability to break out of the DOM tree i.e. we can render a component to a DOM node that is not under this **root** element.
* ReactDOM.createPortal()
* First parameter can be any element that react can render i.e. number, string, JSX, components
* Second parameter where to render.
* ReactDOM.createPortal(‘element-name’)
* This came to picture to deal with CSS of parent component where child components are modals, popups, tool-tips etc.
* React portal can be anywhere in the DOM tree it behaves as normal react child in the every other way.
* **Event bubbling** fired inside a portal will propagate to ancestors containing react tree even if those elements are not ancestors in the DOM tree.

**Error Boundary**

* A class component that implements either one or both of the lifecycle methods **getDerivedStateFromError** or **componentDidCatch** becomes an **error boundary**.
* The static method **getDerivedStateFromError** method is used to render a fallback UI after an error is thrown and **componentDidCatch** method is used to log the error information.
* It can catch error during rendering, lifecycles methods, constructor and whole tree below them, but **can’t handle the error in** **event** **handler**.
* The placement of the error boundary also matters as it controls if the entire app should have fallback UI or just the component causing the problem.

**Higher Order Components**

* HOC is a pattern where a function accepts the component as parameter and returns an enhanced component.
* This is used to share common functionality between the components without repeating the code.
* We have to pass down the remaining props to the wrapped-Component using spread operator.
* We can pass parameters to HOC for different functionality
* We may come across HOC in Redux, Routers, and styles in material-UI.

**Render Props**

* This the other way of share common functionality between the components without repeating the code, it’s similar to HOC.
* The term **Render prop** refers to a technique for **sharing code** between React components using a **prop whose value is a function.**
* In react it is possible to **props** whose value is a **function** to control what is actually render by a component.
* There is a different type of approach which don’t make use of **props** instead **children** **props** are used.

**Context**

* Context provides a way to pass data through the component tree without having to pass props down manually at every level.
* There are **three** steps to implement context.
  + Create the context: using **React.createContext()** and make sure to export provider and consumer.
  + Provide a context value: At the top level include the provider component provide a value attribute, this value can be consumed any of the descendent components.
  + Consume the context value: use the consumer component and pass in function as its child, the function receives context value as its parameter that is used to return desired JSX.
* The place you provide is important because only the descendent components can consume it.
* We can set default value the context, it set while creating context and passed as argument to **creatContext(‘srini’).**
* Context type property on the class.
  + Export default context
  + Assign this context to context type property on the class.
  + It has two limitations
    - It only works with class components.
    - You can only subscribe to single context using contextType.
* Example of having two context types

